

WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases

Ryan Marcus
Brandeis University
ryan@cs.brandeis.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

ABSTRACT

Workload management for cloud databases deals with the tasks of resource provisioning, query placement, and query scheduling in a manner that meets the application’s performance goals while minimizing the cost of using cloud resources. Existing solutions have approached these three challenges in isolation while aiming to optimize a single performance metric. In this paper, we introduce WiSeDB, a learning-based framework for generating *holistic* workload management solutions customized to application-defined performance goals and workload characteristics. Our approach relies on supervised learning to train cost-effective decision tree models for guiding query placement, scheduling, and resource provisioning decisions. Applications can use these models for both batch and online scheduling of incoming workloads. A unique feature of our system is that it can adapt its offline model to stricter/looser performance goals with minimal re-training. This allows us to present to the application alternative workload management strategies that address the typical performance vs. cost trade-off of cloud services. Experimental results show that our approach has very low training overhead while offering low cost strategies for a variety of performance metrics and workload characteristics.

1. INTRODUCTION

Cloud computing has transformed the way data-centric applications are deployed by reducing data processing services to commodities that can be acquired and paid for on-demand. Despite the increased adoption of cloud databases, challenges related to workload management still exist, including provisioning cloud resources (e.g., virtual machines (VMs)), assigning incoming queries to provisioned VMs, and query scheduling within a VM in order to meet performance goals. These tasks strongly depend on application-specific workload characteristics and performance goals, and they are typically addressed by ad-hoc solutions at the application level.

A number of efforts in cloud databases attempt to tackle these challenges (e.g., [8, 9, 16, 18, 19, 20, 27, 29, 30]). However, these techniques suffer from two main limitations. First, they do not provide holistic solutions but instead address only individual aspects of the problem, such as query admission [27, 30], query placement to VMs [9, 18, 19], query scheduling within a VM [8, 9], or VM provisioning [16, 18, 20, 29]. Since these solutions are developed independently of each other, their integration into a unified framework requires substantial effort and investment to “get it right” for each specific case. Second, while a broad range of latency-related performance metrics are covered by these systems (e.g., query response time [8, 9, 19, 29], average query latency [19]), each offers solutions tuned only for a specific metric. Adapting them to support a wider range of application-specific metrics (e.g., max latency, percentile metrics) is not a straightforward task.

Expanding on our vision [21], we argue that cloud-based databases could benefit from a workload management advisor service that removes the burden of the above challenges from application developers. Applications should be able to specify their workload characteristics and performance objectives, and such a service should return a set of low-cost and performance-efficient strategies for executing their workloads on a cloud infrastructure.

We have identified a number of design goals for such an advisor service. First, given an incoming query workload and a performance goal, the service should provide *holistic solutions* for executing a given workload on a cloud database. Each solution should indicate: (a) the cloud resources to be provisioned (e.g., number/type of VMs), (b) the distribution of resources among the workload queries (e.g., which VM will execute a given query), and (c) the execution order of these queries. We refer to these solutions collectively as *workload schedules*.

Second, to support diverse applications (e.g., scientific, financial, business-intelligence, etc), we need a customizable service that supports equally diverse application-defined performance criteria. We envision a *customizable* service that generates workload schedules tuned to performance goals and workload characteristics specified by the application. Supported metrics should capture the performance of individual queries (e.g., query latency) as well as the performance of batch workloads (e.g., max query latency of an incoming query workload).

Third, since cloud providers offer resources for some cost (i.e., price/hour for renting a VM), optimizing schedules for this cost is vital for cloud-based applications. Hence, any workload management advisor should be *cost-aware*. Cost functions are available through contracts between the service providers and their customers in the form of service level agreements (SLAs). These cost functions define the price for renting cloud resources, the performance goals, and the penalty to be paid if the agreed-upon performance is not met. A workload management service should consider *all* these cost factors while assisting applications in exploring performance/cost trade-offs. Since different workload schedules offer different performance vs. cost trade-offs for different performance metrics, the system should be able to discover the “best” strategy for executing a given workload under an application-specific performance goal. Low cost workload schedules should be discovered independently of the performance metric.

This paper introduces *WiSeDB* ([W]orkload management [S]ervice for cloud [DB]s), a workload management advisor for cloud databases designed to satisfy the above requirements. WiSeDB offers customized solutions to the workload management problem by recommending cost-effective strategies for executing incoming workloads for a given application. These strategies are expressed as *decision-tree models* and WiSeDB utilizes a super-

vised learning framework to “learn” models customized to the application’s performance goals and workload characteristics. For an incoming workload, WiSeDB can parse the model to identify the number/type of VMs to provision, the assignment of queries to VMs, and the execution order within each VM, in order to execute the workload and meet the performance objective with low-cost.

Each model is cost-aware: it is trained on a set of performance and cost-related features collected from minimum cost schedules of sample workloads. This cost accounts for resource provisioning as well as any penalties paid due to failure to meet the performance goals. Furthermore, our proposed features are independent from the application’s performance goal and workload specification, which allows WiSeDB to learn effective models for a range of metrics (e.g., average/max latency, percentile-based metrics). Finally, each model is trained offline once, and can be used at runtime to generate schedules for *any* workload matching the model’s workload specifications. Given an incoming batch workload and a decision model, WiSeDB parses the model and returns a low cost schedule for executing the workload on cloud resources.

WiSeDB leverages a trained decision model in two additional ways. First, the training set of the model is re-used to generate a set of *alternative models* for the same workload specification, but stricter or more relaxed performance criteria. Several models are presented to the user, along with the cost estimate for each as function of the workload size. This allows applications to explore the performance vs. cost trade-off for their specific workloads. Second, each model can be adjusted (with small overhead) during runtime to support *online scheduling* of queries arriving one at a time.

The contributions of this work can be summarized as follows:

1. We introduce *WiSeDB*, a workload management advisor for cloud databases. We discuss its design and rich functionality, ranging from recommendations of alternative workload execution strategies that enable exploration of performance vs. cost trade-offs to resource provisioning and query scheduling for both batch and online processing. All recommendations are tuned for application-defined workload characteristics and (query latency-related) performance goals.
2. We propose a novel *learning approach to the workload management problem*. WiSeDB learns its custom strategies by collecting features from optimal (minimum cost) schedules of sample workloads. We rely on a graph-search technique to identify these schedules and generate a training set in a timely fashion. Furthermore, we have identified a set of performance and cost-related features that can be easily extracted from these optimal solutions and allow us to learn effective (low-cost) strategies for executing any workload for various performance metrics.
3. We propose an *adaptive* modeling technique that generates alternative workload execution strategies, allowing applications to explore performance vs. cost trade-offs.
4. We leverage the trained models to schedule batch workloads as well as to support online scheduling by efficiently generating low-cost models upon arrival of a new query.
5. We discuss experiments that demonstrate WiSeDB’s ability to learn low-cost strategies for a number of performance metrics with very small training overhead. These strategies offer effective solutions for both batch and online scheduling independently of workload and performance specifications.

We first discuss WiSeDB’s system model in Section 2 and then define our optimization problem in Section 3. We introduce our modeling framework in Section 4 and the adaptive modeling approach in Section 5. Section 6 discusses WiSeDB’s runtime functionality. Section 7 includes our experimental results. Section 8

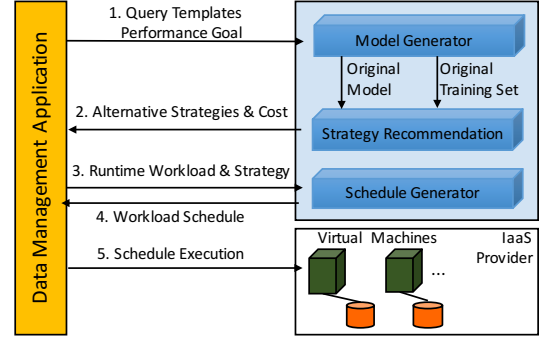


Figure 1: The WiSeDB system model

```
SELECT SUM (l_extendedprice * l_discount)
FROM   lineitem
WHERE  l_shipdate >= date '[DATE]'
AND    l_shipdate < date '[DATE]'
      + interval '1' year
AND    l_discount between [DISC] - 0.01
      and [DISC] + 0.01
AND    l_quantity < [QUANT];
```

Figure 2: TPC-H Query Template #6

discusses related work and we conclude in Section 9.

2. SYSTEM MODEL

Our system is designed for data management applications deployed on an Infrastructure-as-a-Service (IaaS) cloud (e.g., [1,2]). These providers typically offer virtual machines (VMs) of different types (i.e., resource configurations) for a certain fee per renting period. We assume this deployment is realized by renting VMs with preloaded database engines (e.g., PostgreSQL [3]) and that queries can be executed locally on any of the rented VMs. This property is offered by fully replicated databases.¹

Workload Specification Applications begin their interaction with WiSeDB by providing a *workload specification* indicating the query templates (e.g., TPC-H templates [4]) that will compose their workloads. In analytical applications, incoming queries are generally instances of a small number of templates, and queries of the same template have similar performance properties (i.e., latency) because they access and join the same tables [25]. In practice, WiSeDB’s approach is agnostic to the tables accessed or joined and cares only about the latency of each template, i.e., queries with identical latency can be treated as instances of the same template.

Query Templates We assume that a user’s workload is composed of queries, each of which are derived from a finite set of *query templates*, much like TPC-H templates [4]. A query template, generally expressed in SQL, is a query with a set of missing values. Users can instantiate a new query of a given template by providing the missing values. Figure 2 shows an example of a query template from the TPC-H benchmarking suite. A new instance of this query template could be created by giving values for [DATE], [DISC], and [QUANT]. Conversely, any query matching the semantics of some template for a certain set of values can be said to be an instance of that template.

¹Partial replication/data partitioning models can also be supported by specifying which data partitions can serve a given query.

Performance Goals Applications also specify their performance goals for their workloads as functions of query latency. Performance goals can be defined either at the query template level or workload level. Currently, we support the following four types of metrics. (1) *Per Query Deadline*: users can specify an upper latency bound for each query template (i.e., queries of the same template have the same deadline). (2) *Max Latency Deadline*: the user expresses an upper bound on the worst query response time in a query workload. (3) *Average Deadline*: sets an upper limit on the average query latency of a workload. (4) *Percentile Deadline*: specifies that at least $x\%$ of the workload's queries must be completed within t seconds. These metrics cover a range of performance goals typically used for database applications (e.g., [8, 19, 27, 30]). However, WiSeDB is the first system to support them within a single workload management framework.

Performance goals are expressed as part of a Service-Level-Agreement (SLA) between the IaaS provider and the application that states (a) the workload specification, (b) the performance goal, and (c) a penalty function that defines the penalty to be paid to the application if that goal is not met. Our system is agnostic to the details of the penalty function, incorporating it into its cost-model as a “black box” function that maps performance to a penalty amount.

The architecture of the WiSeDB Advisor is shown in Figure 1. Workload and performance specifications are submitted to WiSeDB, which trains a decision model, a.k.a. *strategy* (*Model Generator*). The training set for this model is also leveraged to generate alternative decision models (strategies) for stricter and more relaxed performance goals (*Strategy Recommendation*). These strategies are presented to the user along with a cost function that estimates the monetary cost of each strategy based on the frequency of each query template in a given workload.

Given an incoming workload at runtime, the application estimates the expected cost and performance of executing these workloads using our proposed strategies and chooses the one that better balances performance needs and budget constraints (*Execution Strategy*). WiSeDB identifies (a) the type/number of VMs to be provisioned, (b) the assignment of queries to these VMs and (c) the execution order of the queries within each VM, in order to execute the incoming workload based on the chosen strategy. This step is executed by the *Schedule Generator* which can generate these *workload schedules* for both batch workloads as well as single queries as they arrive (online scheduling). Applications execute their workloads according to WiSeDB's recommendations. They rent VMs as needed and add queries to the processing queue of VMs according to the proposed schedule. VMs are released upon completion of the workload's execution. In the case of online processing, the application adds the new query to the processing queue of an existing VM or to a new VM.

3. PROBLEM DEFINITION

Here, we formally define our system's optimization goal and discuss the problem's complexity. Applications provide a set of *query templates* $T = \{T_1, T_2, \dots\}$ as the workload specification and a *performance goal* R . Given the set of query templates T , WiSeDB generates decision models for scheduling workloads with queries instances drawn from T . Let us assume a workload $Q = \{q_1^x, q_2^y, \dots\}$ where each query $q_i^j \in Q$ is an instance of the template $T_j \in T$. Given a workload Q and one of the generated decision models, WiSeDB identifies a *schedule* S for executing Q .

We represent a VM of type i , vm^i , as a queue $vm^i = [q_1^x, q_2^y, \dots]$ of queries to process in that order and we represent a *schedule* $S = \{vm_1^i, vm_2^j, \dots\}$ of the workload Q as a list of VMs such that each VM contains only queries from Q . Hence,

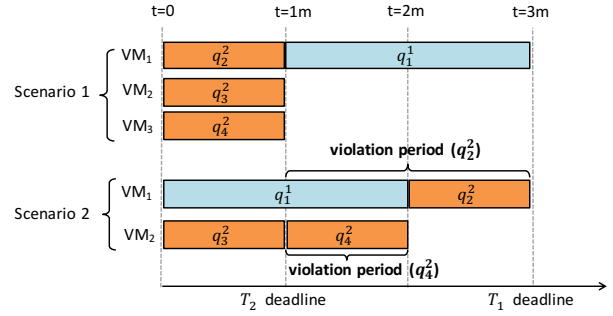


Figure 3: Two different schedules for $Q = \{q_1^1, q_2^2, q_3^2, q_4^2\}$

each schedule S indicates (1) the number and type of VMs to be provisioned, (2) the assignment of each query $q_j^i \in Q$ to these VMs and (3) the query execution order on each VM, $vm_j^i \in S$. A *complete* schedule assigns each query in Q to one VM.

We denote the latency of a query q_j^x (of template T_x) when executed on a VM of type i , vm^i , as $l(q_j^x, i)$. Latency estimates can be provided by either the application (e.g., by executing representative queries a-priori on the available VM types) or by using existing prediction models (e.g., [10, 11]). Queries assigned to a VM can be executed immediately or can be placed in a VM's processing queue if no more concurrent queries can be processed.²

Figure 3 shows two possible schedules for a workload of four queries drawn out of two templates. The first scenario uses three VMs while the second executes the queries on two VMs. Based on our notation, the first schedule is expressed as $S_1 = \{vm_1 = [q_2^2, q_1^1], vm_2 = [q_3^2], vm_3 = [q_4^2]\}$, where as the second scenario is expressed as $S_2 = \{vm_1 = [q_1^1, q_2^2], vm_2 = [q_3^2, q_4^2]\}$. Here, $p(R, S_2) \neq 0$, since the violation period for the schedule represented in the second scenario is not zero.

Cost Model To calculate the monetary cost of processing a workload, we assume each VM of type i has a fixed start-up cost f_s^i , as well as a running cost f_r^i per unit of time (i.e. the price for renting the VM for that time unit). We also assume a penalty function $p(R, S)$ that estimates the penalty for a given schedule S and performance goal R . Without loss of generality (and similarly to the model used by IaaS providers [1]), we assume penalties are calculated based on the violation period, i.e., a fixed amount per time period of violation within a schedule S .

The *violation period* is the duration of time that the performance goal R was not met. If the goal is to complete queries of a given template within a certain deadline (per query deadline goal), the violation period for each query is measured from the time point it missed its deadline until its completion. Figure 3 shows an example of this case. Let us assume that queries of template T_1 have an execution time of 2 minutes and queries of T_2 execute with 1 minute. The figure shows the violation periods for the second scenario (for q_2^2, q_4^2) assuming that the deadline of template T_1 is 3 minute and for T_2 is 1 minutes (the first scenario does not have any violations).

For the maximum latency metric, where no query can exceed the maximum latency, the violation period is computed in the same way. For an average latency performance goal, the duration of the violation period is the difference between the desired average latency and the actual average latency of each query. For a percentile performance goal that requires $x\%$ of queries to be complete in y minutes, the violation period is the amount of time in which $100 - x\%$ of the queries had latencies exceeding y minutes. We

²Most DBMSs put an upper limit on the number of concurrent queries, referred to as multiprogramming level.

Symbol	Meaning
R	performance goal
S	workload schedule
T_i	query template i
q_j^i	j -th query (of template i)
f_s^i	start-up cost of VM type i
f_r^i	cost of renting VM type i per unit time
$l(q^j, i)$	latency of query template j on VM type i
$p(R, S)$	penalty of S under R
$cost(R, S)$	total cost of S under R
vm_j^i	j th VM (of type i)
vm^i	VM of type i
v_s	partial schedule at vertex v
v_u	unassigned queries at vertex v

Table 1: Notation table

denote the monetary cost of performance violations for a given performance goal R and a schedule S as $p(R, S)$.

Problem Definition Given a workload $Q = \{q_1^x, q_2^y, \dots\}$, where q_i^j is of template T_j , and a performance goal R , our goal is to find a complete schedule S that minimizes the total monetary cost (provisioning, processing, and penalty payouts costs) of executing the workload. We define this total cost, $cost(R, S)$, as:

$$cost(R, S) = \sum_{vm_j^i \in S} \left[f_s^i + \sum_{q_k^m \in vm_j^i} f_r^i \times l(q_k^m, i) \right] + p(R, S) \quad (1)$$

Problem Complexity Under certain conditions, our optimization problem becomes the bin packing problem, where we try to “pack” each query into one of the available “VM-bins”. For this reduction, we need to assume that (1) the number of query templates is unbounded, (2) infinite penalty, $p(R, S) = \infty$, if the performance goal is violated, and (3) the start-up cost f_s^i is uniform across all VM types. Under these conditions, the problem is NP-Hard. However, these assumptions are not valid in our system. Limiting the number of query templates relaxes the problem to one of polynomial complexity, but still not computationally feasible [28].

Two common greedy approximations to this optimization problem are the first-fit decreasing (FFD) [22] and first-fit-increasing (FFI) algorithms, which sort queries in decreasing or increasing order of latency respectively and place each query on the first VM where the query “fits” (incurs no penalty). If the query will not fit on any VM, a new VM is created. Existing cloud management systems have used FFD (e.g., [7]) for provisioning resources and scheduling queries. However, it is often not clear which of these greedy approaches is the best for a specific workload and performance goal. For example, when applied to the workload and performance goals shown in Figure 3, FFD schedules all queries on their own VM, which offers the same performance as scenario 1 but uses an additional VM (and hence has higher cost). A better approximation would be FFI, which produces the schedule that is depicted in scenario 1, scheduling the four queries across three VMs without violating the performance goal.

Furthermore, there might be scenarios when none of these approximations offer the best solution. For example, consider workloads consisting of three templates, T_1, T_2, T_3 with latencies of four, three, and two minutes respectively. Assume we have two queries of each template, q_1^1, q_2^1 of template T_1 , q_3^2, q_4^2 of template T_2 , and q_5^3, q_6^3 of template T_3 and we wish to keep the total execution time of the workload below nine minutes. FFD will find the schedule $S_{FFD} = \{[q_1^1, q_2^1], [q_3^2, q_4^2, q_5^3], [q_6^3]\}$, while FFI would

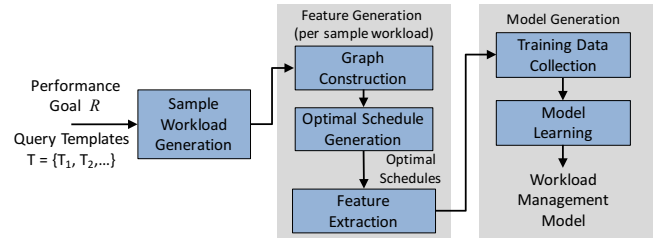


Figure 4: Generation of the decision model

find the schedule $S_{FFI} = \{q_5^3, q_6^3, q_3^2\}, [q_4^2, q_1^1], [q_2^1]\}$. However, a better strategy is one that attempts to place an instance of T_1 , then an instance of T_2 , then an instance of T_3 , then create a new VM, resulting in the schedule $S' = \{[q_1^1, q_3^2, q_5^3], [q_2^1, q_4^2, q_6^3]\}$, which has a lower cost because it provisions one less VM.

WiSeDB departs from the “one-strategy-fits-all” approach used by standard approximation heuristics. Instead, it offers effective scheduling strategies for custom performance goals and workload specifications by learning heuristics tailored to the application’s needs. We next describe how WiSeDB identifies such strategies.

4. DECISION MODEL GENERATION

WiSeDB relies on supervised learning to address our workload management problem. Next, we describe this process in detail.

4.1 Approach Overview

Given an application-defined workload specification (i.e., query templates and a performance goal), WiSeDB generates a set of workload execution strategies that can be used to execute incoming query workloads with low cost and within the application’s performance goal. Formally, our goal is to identify strategies that minimize the total cost as defined in Equation 1. Towards this end, our framework generates samples of optimal schedules (i.e., that minimize the total cost) and relies on decision tree classifiers to learn “good” strategies from these optimal solutions.

Our framework is depicted in Figure 4. Initially, we create a large number of *random sample workloads*, each consisting of a small number of queries drawn from the query template definitions. Our next step is to identify the optimal schedule for each of these sample workloads. To do this efficiently, we represent the problem of scheduling workloads as a *graph navigation* problem. On this graph, edges represent query assignment or resource provisioning decisions and the weight of each edge is equal to the cost of that decision. Hence, each path through the graph represents decisions that compose some schedule for the given workload. Finding the optimal schedule for that workload is thus reduced to finding the shortest path on this graph.

Next, for each decision within an optimal schedule, we extract a set of features that characterize this decision. We then generate a training set which includes all collected features from all the optimal schedules across all sample workloads. Finally, we train a *decision tree model* on this training set. The learning process is executed offline and the generated models can be used during runtime on incoming workloads. Next, we describe these steps in detail.

4.2 Workload Sampling

WiSeDB first generates sample workloads based on the application-provided query templates T . We create N random sample workloads, each containing m queries. Here, N and m must be sufficiently large so that query interaction patterns emerge and the decision tree model can be properly trained. However, m must also be sufficiently small so that for each sample workload we can identify the optimal schedule in a timely manner.

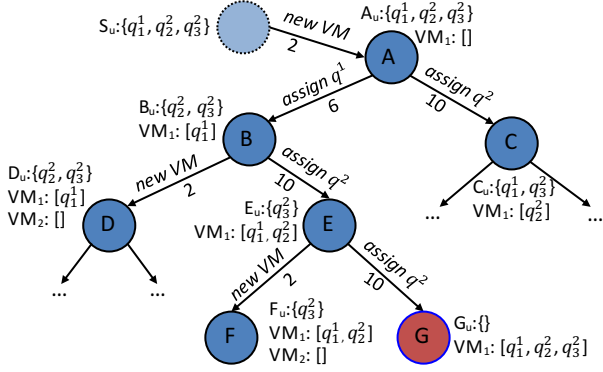


Figure 5: A subgraph of a scheduling graph for two query templates and $Q = \{q_1^1, q_2^2, q_3^3\}$. G is one goal vertex

In order to ensure that our sampling covers the space of possible workloads, we rely on uniform direct sampling of the query templates. If the sampling is not uniform, the decision tree may not be able to learn how two query templates interact, or the decision tree may have very little information about certain templates. Furthermore, we generate a large number of samples (e.g., in our experiments $N = 3000$) in order to ensure that our workload samples will also include workloads that are imbalanced with respect to the number of unique templates they include. This allows WiSeDB to handle skewed workloads.

Even though our training will be based on uniform sampling of the query templates, WiSeDB can still handle skewed workloads. This is because uniform direct sampling generates some sample workloads that are balanced and some sample workloads that are not balanced. The unbalanced sample workloads teach the decision tree classifier how to handle odd, skewed workloads, and the balanced workloads teach the decision tree how to handle the “usual” mixes. We demonstrate this experimentally in Section 7.5.

4.3 Optimal Schedule Generation

Given a set of sample workloads, WiSeDB learns a model based on the optimal schedules for these workloads. To produce these optimal solutions, we represent schedules as paths on a weighted graph, and we find the minimum cost path on this graph. This graph-based approach provides a number of advantages. First, each “best” path represents not only an optimal schedule, but also *the steps taken to reach that optimal schedule*. The information relevant to each optimal decision is captured by each vertex’s state. Second, a graph representation lends itself to a natural way of *eliminating redundancy in the search space* via careful, strategic path pruning. Finally, the well-studied nature of *shortest-path problems* enables the application of deeply-understood algorithms with desirable properties. Next, we describe our graph construction in detail, and highlight these advantages.

Graph Construction Given a sample workload $Q = \{q_1^1, q_2^2, \dots\}$, we construct a directed, weighted graph $G(V, E)$ where vertices represent intermediate steps in schedule generation, i.e., partial schedules and a set of remaining queries to be scheduled. Edges represent actions, such as renting a new VM or assigning a query to a VM. The cost (weight) of each edge will be the cost of performing a particular action (e.g., the cost of starting a new VM). We refer to this as a *scheduling graph*.

Formally, each vertex $v \in V$ has a schedule for some queries of the workload, $v_s = \{vm_1^i, vm_2^k, \dots\}$, which includes the VMs to be rented for these queries. Each VM j of type i , vm_j^i , is a queue of queries that will be processed on that VM, $vm_j^i = [q_k^x, q_m^y, \dots]$.

Hence, v_s represents possible (potentially partial) schedules for the workload Q . Each v also has a set of unassigned queries from Q , v_u , that must still be placed onto VMs.

The *start vertex*, $A \in V$, represents the initial state where all queries are unassigned. Therefore, A_u includes all the queries in the given workload and A_s is empty. If a vertex $g \in V$ has no unassigned queries, we say that vertex g is a *goal vertex* and its schedule g_s is a *complete schedule*.

An edge in E represents one of two possible actions:

1. A *start-up edge* (u, v, i) represents renting a new VM of type i , vm_j^i . It connects a vertex u to v where v has an additional empty VM, i.e., $v_s = u_s \cup vm_j^i$. It does not assign any queries, so $u_u = v_u$. The weight of a start-up edge is the cost to provision a new VM of type i : $w(u, v) = f_s^i$.
2. A *placement edge* (u, v, q_y^x) represents placing an unassigned query $q_y^x \in u_u$ into the queue of a rented VM in v_s . It follows that $v_u = u_u - q_y^x$. Because WiSeDB is agnostic to the specifics of any particular query, the placement of an instance of query template T_x is equivalent to placing any other instance of T_x . Therefore, we include only a single placement edge per query template even if the template appears more than once in u_u . The cost of an edge that places query q_y^x into a VM of type i is the execution time of the query multiplied by the cost per unit time of the VM, plus any *additionally* incurred penalties:

$$w(u, v) = \left[l(q_y^x, i) \times f_r^i \right] + [p(R, v_s) - p(R, u_s)] \quad (2)$$

Figure 5 shows part of a scheduling graph for a workload $Q = \{q_1^1, q_2^2, q_3^3\}$. A represents the start vertex and G represents a goal vertex.³ At each edge, a query is assigned to an existing VM (\overline{AB} , \overline{AC} , \overline{BE} , \overline{EG}), or a new VM is created (\overline{BD} , \overline{EF}). The path \overline{ABEG} represents assigning the three queries, q_1^1, q_2^2, q_3^3 , to be executed in that order on vm_1 .

The weight of a path from the start vertex A to a goal vertex g will be equal to the cost of the complete schedule of the goal vertex, $cost(R, g_s)$, for a given performance goal R . Since all complete schedules are represented by some goal state, searching for a minimum cost path from the start vertex A to any goal vertex g will provide an optimal schedule for the workload Q . We then find the optimal path through the graph, noting the scheduling decision made at every step, i.e., which edge was selected at each vertex.

Graph Reduction To improve the runtime of the search algorithm, we reduce the graph in a number of ways. First, we include a start-up edge only if the last VM provisioned has some queries assigned to it, i.e., we allow renting a new VM *only if the most recently provisioned VM is not empty*. This eliminates paths that provision VMs that are never used. Second, queries are assigned only to the most recently provisioned VM, i.e., each vertex has outgoing placement edges that assign a query *only to the most recently added VM*. This reduces the number of redundant paths in the graph, since each combination of VM types and query orderings is accessible by only a single instead of many paths. This reduction can be applied without loss of optimality, e.g. without eliminating any goal vertices.

LEMMA 4.1. *Given a scheduling graph G and a reduced scheduling graph G_r , all goal vertices with no empty VMs in G are reachable in G_r .*

PROOF. Consider an arbitrary goal vertex with no empty VMs $g \in G$. For any vertex v , let $head(v_s)$ be the most recently created VM in v_s (the head of the queue).

³The dotted vertex represents the provisioning of the first VM which is always the first decision in any schedule.

Let us assume $head(g_s) = vm_j^i$ and q_y^x is the last query scheduled in vm_j^i . Then there is a placement edge $e_p = (v, g, q_y^x)$ connecting some vertex v with query q_y^x in its set of unassigned queries, i.e., $q_y^x \in v_u$, to g . Further, we know that e_p is an edge of G_r because e_p is an assignment of a query to the most recently created VM. If $head(v_s)$ is non-empty, and q_k^m is the last query on $head(v_s)$, then there is another vertex u connected to v via a placement edge $e_p = (u, v, q_k^m)$ in G_r by the same argument. If $head(v_s) = vm_j^i$ is empty, then there is a start-up edge $e_s = (y, v, i)$ connecting some vertex y , with the same set of unassigned queries as v , i.e., $v_u = y_u$, to v . We know that e_s must be an edge of G_r because v can have at most one empty VM. This process can be repeated until the start vertex is reached. Therefore, there is a path from the start vertex in G_r to any goal vertex with no empty VMs $g \in G$. \square

Search Heuristic WiSeDB searches for the minimum cost path (i.e., optimal schedule) from the start vertex to a goal vertex using the A* search algorithm [14] which offers a number of advantages. First, it is complete, meaning that it always finds an optimal solution if one exists. Second, A* can take advantage of an *admissible heuristic*, $h(v)$, to find the optimal path faster. An admissible $h(v)$ provides an estimate of the cost from a vertex v to the optimal goal vertex that is always less or equal to the actual cost, i.e., it must never overestimate the cost. For any given admissible heuristic, A* is optimally efficient, i.e., no other complete search algorithm could search fewer vertices with the same heuristic.

The heuristic function is problem specific: in our system, the cost of a path to v is the cost of the schedule in v_s , $cost(R, v_s)$, thus it is calculated differently for different performance goals R . Hence satisfying the admissibility requirement depends on the semantics of the performance metric. Here, we distinguish *monotonically increasing* performance metrics from those that are not. A performance goal is monotonically increasing if and only if the penalty incurred by a schedule u_s never decreases when adding queries. Formally, at any assignment edge connecting u to v , $p(R, v_s) \geq p(R, u_s)$. Maximum query latency is monotonically increasing performance metric, since adding an additional query on the queue of the last provisioned VM will never decrease the penalty. Average latency is not monotonically increasing, as adding a short query may decrease the average latency and thus the penalty. For monotonically increasing performance goals, we define a heuristic function $h(v)$ that calculates the cheapest possible runtime for the queries that are still unassigned at v . In other words, $h(v)$ sums up the cost of the cheapest way to process each remaining query by assuming VMs could be created for free⁴:

$$h(v) = \sum_{q_y^x \in v_u} \min_i \left[f_r^i \times l(q_y^x, i) \right] \quad (3)$$

LEMMA 4.2. *For monotonically increasing performance goals, the search heuristic defined in Equation 3 is admissible.*

PROOF. Regardless of performance goals, query classes, or VM performance, one always has to pay the cost of renting a VM for the duration of the queries. More formally, every path from an arbitrary vertex v to a goal vertex must include an assignment edge, with cost given by Equation 2, for each query in v_u and for some VM type i . When the performance goal is monotonically increasing, the term $p(R, v_s) - p(R, u_s)$ is never negative, so $h(v)$ is never larger than the actual cost to the goal vertex. \square

⁴For performance goals that are not monotonically increasing, we do not use a heuristic, which is equivalent to using the null heuristic, $h(v) = 0$.

4.4 Feature Extraction

After we have generated the optimal schedules for each of the sampled workloads, we generate the training set for our decision tree classifier. The training set consists of (*decision, features*) pairs indicating the decisions that was made by A* while calculating optimal schedules and performance/workload related features at the time of the decision. Each decision represents an edge in the search graph, and is therefore a decision to either (a) create a new VM of type i , or (b) assign a query of template T_j to the most recently created VM. Thus, the domain of possible decisions is equal to the sum of the number of query templates and the number of VM types.

We map each decision (edge) (u, v) in the optimal path to a set of features of its origin vertex u since there is a correspondence between a vertex and the optimal decision made at that vertex. Specifically, for a given vertex u , the edge selected in the optimal path is independent of u 's parents or children but depends on the unassigned queries u_u and the schedule so far, u_s . However, the domains of u_u and u_s are far too large to enumerate and do not lend themselves to machine learning algorithms (u_u and u_s are neither numeric or categorical). Therefore, we extract features from *each* of the vertices in *all* the optimal paths we collected for *all* of the N sample workloads.

Feature Selection One of the main challenges of our framework is to identify a set of efficiently-extracted features that can lead to a highly effective decision model. Here, the space of candidate features is extremely large, as one could extract features relating to queries (e.g., latency, tables, join, predicates, etc.), the underlying VM resources (CPU, disk, etc.), or combinations thereof. Since even enumerating all possible features would be computationally difficult, finding the optimal set of features is not feasible. Therefore, we focus on features that (1) are fast to extract, (2) capture performance and cost factors that may affect the scheduling decisions, and (3) have appeared in existing workload scheduling and resource provisioning heuristics [7, 8, 19].

Surprisingly, many “common sense” features did not prove to be effective. For example, we extracted the number of available queries of a certain template X still unassigned, *unassigned-X*. Since each training sample is small, the training set only contained small values of *unassigned-X*, so the decision tree did not learn how to handle the very large values of *unassigned-X* encountered at runtime. We find that effective features must be *unrelated to the size of the workload*, since the large workload sizes encountered at runtime will not be represented in the training set. Additionally, we extracted the number of assigned queries of a certain template X , *assigned-X*. With tight performance goals, this feature is well-represented in the training data because machines “fill up” after a small number of queries. However, this feature shares information with *wait-time*, as the two are highly related. Because of this redundancy, adding *assigned-X* to the model did not improve performance. Therefore, effective features must be reasonably *independent from one another* to avoid representing redundant data.

We have experimentally studied numerous features which helped us form a set of requirements for the final feature set. First, our selected features should be agnostic to the specifics of the query templates and performance goals. This will allow our framework to be customizable and enable it to learn good strategies independently of the query templates and performance metric expressed by the application.

Second, effective features must be unrelated to the size of the workload, since the large workload sizes encountered at runtime will not be represented in the training sample workloads, whose

size is restricted in order to obtain the optimal schedules in a timely manner (e.g. tracking the exact number of unassigned instances of a particular template will not be useful, as this value will be very small during training and very large at runtime).

Third, features must be independent from one another to avoid extracting redundant information. For example, the wait time in a given VM is related to the number of queries assigned to that VM, and we observed that including only one of these metrics in our feature list was sufficient to give us low-cost schedules.

From an information-theoretic point of view, the minimum number of bits needed for an optimal feature set is $\log_2(|T| + |V|)$, where $|T| + |V|$ is sum of the number of query templates and the number of VM types, because this is the minimum number of bits needed to differentiate each edge in the graph. However, extracting such an optimal set of features is not likely to be computationally-efficient because of the complexity of the problem. Since we require feature extraction to be fast, we do not attempt to find a feature set with such low entropy. Computationally-efficient approaches must therefore make use of higher-entropy, easy-to-extract features as guides towards good approximate solutions.

Based on these observations, we extract the following features for each vertex v along the optimal path:

1. **wait-time**: the amount of time that a query would have to wait before being processed if it were placed on the most recently created VM. Formally, **wait-time** is equal to the execution time of all the queries already assigned to the last VM. This feature can help our model decide which queries should be placed on the last added VM based on their deadline. For example, if a machine's wait time has exceeded the deadline of a query template, it is likely that no more queries of this template should be assigned to that VM. Alternatively, if a machine's wait time is very high, it is likely that only short queries should be assigned.
2. **proportion-of-X**: the proportion of queries on the most recently created VM that are of query template X . In other words, **proportion-of-X** is the ratio between the number queries of template X assigned to the VM and the total number of queries assigned to the VM. For example, if the VM currently has four queries assigned to it, with one of those queries being of template T_1 and three being of template T_2 , then we extract the features **proportion-of-T1**=0.25 and **proportion-of-T2**=0.75. We only need to consider the most recently created VM because the assignment edges in the reduced graph only assign queries to the most recent VM. Since each sample workload contains only a limited number of queries, keeping track of the exact number of instances of each query template would not scale to large workloads. Therefore, we track the proportion instead.
3. **supports-X**: whether or not the most recently created VM is capable of processing a query of class X . This feature is needed when not all VM types are capable of handling every type of query.
4. **cost-of-X**: the cost incurred (including any penalties) by placing a query of template X on the most recently created VM, or infinity if the most recently created VM does not support queries of class X . **cost-of-X** is equal to the weight of the outgoing assignment edge for template X . This allows our model to check the cost of placing an instance of a certain query template and, based on its value, decide whether to assign another query to the last rented VM or create a new VM.

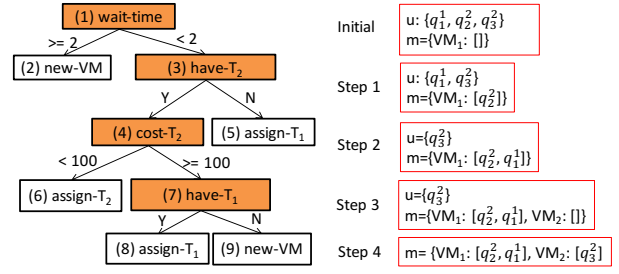


Figure 6: An example decision model

5. **have-X**: whether or not a query of template X is still unassigned. This feature helps our model understand how the templates of the unassigned queries affects the decisions on the optimal path. If there is no instance of some query template T_j unassigned, then the model places one of the remaining templates. If an instance of T_j exists, the model might prefer to schedule that as opposed to any other template.

WiSeDB learns models that combine these features into custom-tailored heuristics for specific workloads and performance goals.

We note that while these features are not enough to *uniquely* identify a vertex and thus learn the *exact* conditions that lead to the optimal schedule, they can shed light on the workload/performance conditions related to the optimal decision. Furthermore, although we cannot claim that these features will *always* allow WiSeDB to learn effective heuristics, our experimental results indicate that these features allow WiSeDB to learn a reasonable cross-section of the scheduling algorithm space, and that they are expressive enough to generate scheduling strategies capable of efficiently handling commonly used performance goals. One can invent an adversarial performance goal (for example, penalizing all VMs without a prime number of assigned queries) that will fall outside the scope of this feature set.

4.5 Workload Management Model

Given a training set, WiSeDB uses a decision tree learner to generate a workload management model. Figure 6 shows an example model defined for two query templates T_1 and T_2 . Each feature node (orange node) of the decision tree represents either a binary split on a numeric or boolean feature. The decision nodes (white nodes) represent the suggested actions.

The decision tree maps a vertex v to an action a by extracting features from v and descending through the tree. Since each action a is represented by a single out-edge of v , one can navigate the decision tree by traversing the edge corresponding to the action selected.

The right side of Figure 6 shows how the decision tree is used to come up with the schedule for a workload $Q = \{q_1^1, q_2^2, q_3^2\}$. Each query in T_1 has a latency of two minutes and the goal is to execute it within three minutes. Instances of T_2 have a latency of one minute and the goal is for each instance to be completed within one minute. For simplicity, we assume VMs of a single type and that queries are executed in isolation.

Given this workload, the tree is parsed as follows. In the first node (1), we check the wait time, which is zero since all queries are unassigned, hence we proceed to node (3). The workload has queries of template T_2 and therefore we proceed to node (4). Here we calculate the cost of placing an instance of T_2 . Let us assume the cost is less than 100 (no penalty is incurred), which leads to node (6) which assigns an instance of T_2 to the first VM. Since we have more queries in the workload we next re-parse the decision tree. In node (1) we check the wait time on the most recent VM which is now 1 minute (the runtime of queries of T_2) so we move to node

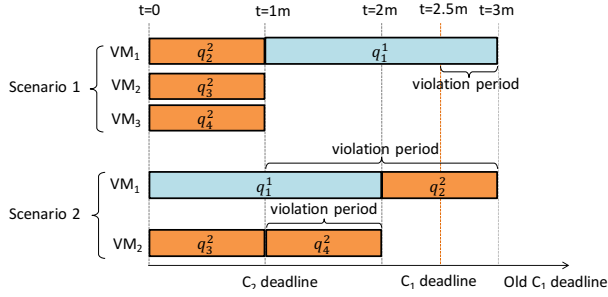


Figure 7: Two different schedules with shifted SLA

(3). Since we have one more query of T_2 unassigned, we move to (4). Now the cost of assigning a query of T_2 is more than 100 since the new query would need to wait for q_1^1 to complete (and thus incur a penalty). Hence, we move to node (7) and we check if there are any unassigned instances of T_1 . Since there are (q_1^1), we assign q_1^1 to the last VM. We re-parse the tree in the same way and by following nodes (1)→(2), then again as (1)→(3)→(4)→(7)→(9), so we assign the remaining query q_3^2 onto a new VM.

Each model represents a workload scheduling strategy. Given a batch of queries, the model in Figure 6 will place an instance of T_2 , then an instance of T_1 , and then create a new VM. This process will repeat itself until queries of T_1 or T_2 are depleted from the incoming batch. When all instances from one of the templates are assigned, single instances of the remaining template will be placed on new VMs until none remain. This strategy is equivalent to first-fit increasing, which sorts queries in decreasing order of latency and places each query on the first VM where the query “fits” (i.e., incurs no penalty).

5. ADAPTIVE MODELING

It is often desirable to allow users to explore performance/cost trade-offs within the space of possible performance goals [24]. This can be achieved by generating different models for the same workload with stricter/looser performance goals and thus higher/lower costs. However, WiSeDB tunes its decision model for a specific goal. Changes in this goal will trigger WiSeDB to re-train the model, since changes in performance goal lead to different optimal schedules and hence different training sets. Therefore, generating a set of alternative decision models for numerous performance constraints could impose significant training overhead.

To address this challenge, WiSeDB employs a technique that adapts an existing model trained for a given workload and performance goal to a new model for the same workload and stricter performance goals. Our *adaptive modeling* requires little re-training since it leverages the fact that two decision models will share significant parts of their training sets if they were trained for the same query templates T and similar performance goals. Our approach relies on the adaptive A* algorithm [17], which reuses information from one graph to create a new search heuristic, $h'(v)$, to search another graph with identical structure but increased edge weights.

Graph Reuse Let us assume a model α trained for templates T and goal R and a desired new model α' trained for the same templates T but a different performance goal R' . Without loss of generality, we only consider cases where the performance goal R' is stricter than R , since one can start with a substantially loose performance goal than the one requested and restrict it incrementally.

For example, Figure 7 shows a modified version of the SLA shown in Figure 3. In Figure 7, the deadline for query class C_2

has been moved up by half a minute, causing Schedule 1 to incur a penalty it did not previously incur.

In order to generate a new model α' , WiSeDB re-uses the training data of the existing model α as follows. For each sample workload used for α , it modifies its corresponding scheduling graph G by updating the weights to reflect the new performance goal R' . Specifically, start-up edges maintain the same cost (the cost to start a new VM), while the cost of placement edges increase due to a stricter performance goals and hence higher penalty fees. Formally, the new cost of an assignment edge (u, v, q_y^x) that places an unassigned query $q_y^x \in u_u$ into the queue of a rented VM in v_s is:

$$w(u, v) + [p(R', v_s) - p(R, v_s)] - [p(R', u_s) - p(R, u_s)]$$

To identify the minimum cost path on the updated graph, we use a new heuristic. For metrics that are monotonically increasing, the new heuristic h' is expressed in terms of the original one h as:

$$h'(v) = \max [h(v), \text{cost}(R, g) - \text{cost}(R, v)]$$

For performance goals that are not monotonically increasing, like average latency, we simply drop the $h(v)$ term, giving $h'(v) = \text{cost}(R, g) - \text{cost}(R, v)$.

We use h' to find the optimal schedule for each of α 's sample workloads (i.e., we search for the minimum cost paths on each sample's scheduling graph G with the updated weights). These new paths will serve as training samples for α' . Intuitively, $h'(v)$ gives the cost of getting from vertex v to the optimal goal vertex under the old performance goal R . Since the new performance goal is strictly tighter than R , $h'(v)$ cannot overestimate the cost, making it an admissible heuristic. Next, we prove this formally.

LEMMA 5.1. $h'(v)$ is an admissible heuristic, i.e., it does not overestimate the cost of the schedule v_s at the vertex v .

PROOF. Based on our cost formula (Equation 1), for any schedule s generated by the model α , the new performance goal R' will only affect the penalty (i.e., since stricter performance goals on the same schedule can lead only to more violations). Therefore:

$$\forall s(p(R', s) \geq p(R, s)) \quad (4)$$

To map this into our scheduling graph, let us consider the set of vertices along an optimal path to a goal vertex g for one sample workload discovered used for the training of α . For any particular vertex v , we know from Equations 1 and 4 that $\text{cost}(R', v) \geq \text{cost}(R, v)$. In other words, the cost of that vertex's partial schedule will cost more if we have a stricter performance goal. Since this holds for all vertices on the optimal path, it also holds for the optimal goal vertex g under α . Hence, if the performance goal becomes stricter, the cost of the final schedule can only increase.

This is illustrated in Figure 7: the cost of Scenario 1 increased when the performance goal was shifted.

Furthermore, for any vertex v along the optimal path to g , we know that the minimum possible cost to go from v to g is exactly $\text{cost}(R, g) - \text{cost}(R, v)$. If the edge weight can only increase due to a stricter performance goal, then the cost to go from v to g under R' must be less than or equal to the cost to go from v to g under R . Formally, since $\text{cost}(R', v) \geq \text{cost}(R, v)$, it follows that $\text{cost}(R', g) - \text{cost}(R', v) \geq \text{cost}(R, g) - \text{cost}(R, v)$.

For example, in the graph shown in Figure 5, $\text{cost}(R, B) = 6$ and $\text{cost}(R, G) = 26$. The cost to get from B to G is thus 20. If R' resulted in the assignment \overline{EG} having a higher cost of 100 (because q_3^2 misses its deadline), then it holds that the cost of going from B to G under R' (110) is greater than the cost of going from B to G under R : $\text{cost}(R', G) - \text{cost}(R', B) \geq \text{cost}(R, G) - \text{cost}(R, B)$.

While the optimal goal vertex for a set of queries may be different under R than under R' , the cost of the optimal vertex g' under R' is no less than the cost of the optimal vertex g under R , because g_s was optimal under R . Intuitively, if there was some vertex γ and complete schedule γ_s of a workload under R' with a lower cost than optimal schedule g_s , then γ_s would also have a lower cost than g_s under R , which contradicts that g_s is optimal. Therefore the cost from any vertex v under R' to the unknown optimal goal vertex g' is no less than the cost to get from v to g under R :

$$\text{cost}(R', g') - \text{cost}(R', v) \geq \text{cost}(R, g) - \text{cost}(R, v)$$

Hence $\text{cost}(R, g) - \text{cost}(R, v)$ is admissible since it never overestimates the actual cost to get from a vertex v to the unknown goal vertex g' under the performance R' . \square

Since $h'(v)$ is admissible, answers produced by using it are guaranteed by the A^* algorithm to be correct [14]. The improved heuristic is able to find the optimal solution much faster, as we will demonstrate experimentally in Section 7. This approach saves WiSeDB from searching the entire scheduling graph for each new model, which is the step with the dominating overhead.

6. RUN TIME FUNCTIONALITY

Using our adaptive modeling approach, WiSeDB recommends to the application a set of decision models (a.k.a. workload management *strategies*) for scheduling incoming queries. During run-time, the user selects a model with a desirable performance vs. cost trade-off. Given a batch query workload, WiSeDB uses the selected model to generate the schedule for that workload. The same model can also be used for online scheduling where queries arrive one at a time. The system remains in this mode until the user either wishes to switch strategy or has no more queries to process.

6.1 Strategy Recommendation

While generating alternative decision models, our goal is to identify a small set of k models that represent significantly different performance vs. cost trade-offs. To achieve this we first create a sequence of performance goals in increasing order of strictness, $\mathcal{R} = R_1, R_2, \dots, R_n$, in which the application-defined goal R is the median. For each goal $R_i \in \mathcal{R}$, we train a decision model (by shifting the original model as described in Section 5) and calculate the average cost of each query template over a large random sample workload Q . Then, we compute the pairwise differences between the average cost of queries per template of each performance goal $R_i \in \mathcal{R}$ using Earth Mover's Distance [6]. We find the smallest such distance between any pairs of performance goals, $EMD(R_i, R_{i+1})$, and we remove R_{i+1} from the sequence. We repeat this process until only k performance goals remain.

A similar technique was also used in [24] where it was shown that it produces a desirable distribution of performance goals (or “service tiers”) that represent performance/cost trade-offs. However, unlike in [24], we do not need to *explicitly* execute any workloads in order to compute the expected cost of a particular strategy. Instead, for each strategy, WiSeDB provides a cost estimation function that takes as a parameter the number of instances per query template. Users can easily experiment with different parameters for incoming workloads and can estimate the expected cost of executing these workloads using each of the proposed strategies.

6.2 Batch Query Scheduling

Given one of the recommended strategies and a batch of queries to be executed on the cloud, WiSeDB uses the strategy and produces a schedule for this workload. A detailed example of this

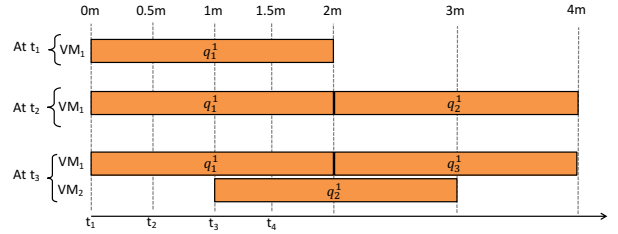


Figure 8: Online scheduling example

process was given in Section 4.5. The schedule indicates the number and types of VMs needed, the assignment of queries to VMs and the query execution order in each VM. The application is then responsible for executing these scheduling decisions on the IaaS cloud infrastructure. In the event that the workload includes a query that cannot be exactly matched to one of the templates given in the workload specification, WiSeDB will treat the unknown query as an instance of the query template with the closest predicted latency. While queries of unseen templates are obviously not ideal, this solution ensures that queries of unseen templates will at least be placed appropriately based on their latency, since two queries with identical latency are identical to WiSeDB.

6.3 Online Query Scheduling

WiSeDB can also handle non-preemptive online query scheduling. We will describe how WiSeDB handles the general case of online scheduling and then we will discuss two optimizations.

General Online Scheduling Let us assume a decision model α trained for a performance goal R and templates T . We also assume that a user submits queries, $\{q_1^x, q_2^y, \dots\}$ at times $\{t_1, t_2, \dots\}$ respectively. Online scheduling can be viewed as a series of successive batch scheduling tasks where each batch includes a single additional query. The first query is scheduled at time t_1 as if it was a batch. When query q_i^x arrives at t_i , we create a new batch B_i containing the queries that have not started executing by t_i and use the model to re-schedule them along with query q_i^x .

Scheduling B_i as a batch using the same decision model might not offer low-cost solutions. This is because this batch includes queries that have been sitting in the queue, and hence their expected latency is now higher than what was assumed when training the model. Formally, at any given time t_i , any query q_y^x that has yet to run will have a wait time of $(t_i - t_y)$ and thus a final latency of $l'(q_y^x, k) = l(q_y^x, k) + (t_i - t_y)$, where k is the type of the VM on which q_y^x was assigned and t_y is the arrival time of q_y^x . To address this, WiSeDB treats q_y^x as if it were of a “new” template which has the same structure at T_x but its expected latency is $l'(q_y^x, k)$. Then, it trains a new decision model for the augmented template set that includes this “new” template. This new model will provide schedules that account for the elapsed wait time of q_y^x .

Figure 8 shows an example. Here, a model α is trained for one template T_1 . Instances of T_1 have a latency of 2m. We assume queries arrive in the order: $\{q_1^1, q_2^1, q_3^1\}$. WiSeDB first schedules query q_1^1 , which arrived at t_1 , as a batch $B_1 = \{q_1^1\}$. At t_2 , q_2^1 arrives and WiSeDB creates a batch $B_2 = \{q_2^1\}$ for it and generates a schedule using model α which assigns it to the first VM, right after the first query. At time t_3 , when one more query q_3^1 arrives, q_2^1 has not yet started executing, so we create a new batch $B_3 = \{q_2^1, q_3^1\}$. However, even though q_2^1 and q_3^1 are of the same template, they have different execution times: q_3^1 has a latency of 1 minute, but q_2^1 has a latency of $1 + (t_3 - t_2)$ minutes since it has been waiting for $(t_3 - t_2)$ minutes. Using model α , which has been trained to handle queries of with an execution time of one minute, might not produce

a desirable (low-cost) schedule for B_3 . So, we train a new model, α' , whose workload specification includes an “extra” template for instances of T_1 with latency of $1 + (t_3 - t_2)$ minutes. α' places q_3^1 after q_1^1 on the first VM and q_2^1 onto a new VM.

6.3.1 Retraining Optimizations

A substantially fast query arrival rate could rapidly create “new templates” as described above, causing a high frequency of retraining. Since training can be expensive, we offer two optimizations to prevent or accelerate retraining.

Model Reuse Upon arrival of a new query, WiSeDB creates a new decision model which take into account the wait times of the queries submitted so far but not already running. WiSeDB strives to reuse models, aiming to reduce the frequency of retraining. In the above example, let us assume a new query q_4^1 arrives at t_4 , at which point we need to generate a schedule for $B_4 = \{q_3^1, q_4^1\}$ (q_1^1 and q_2^1 are running). If $(t_3 - t_2) = (t_4 - t_3)$, then the previously generated model α' can be reused to schedule B_4 , since the “new” template required to schedule q_3^1 is the same as the “new” template previously used to schedule q_2^1 .

Next, let us consider the general case. We define $\omega(i)$ to be the difference in time between the arrival of oldest query that has not started processing at time t_i and the arrival of the newest query at time t_i . Formally,

$$\omega(i) = t_i - t_j \text{ where} \\ j = \min\{j \mid j \leq i \wedge \exists q_y^x (q_y^x \in B_j \wedge q_y^x \in B_i)\}$$

Clearly, the absolute time of a workload’s submission is irrelevant to the model: only the difference between the current time and the time a workload was submitted matters. Formally, if $\omega(i) = \omega(j)$, then B_i and B_j can be scheduled using the same model. In practice, B_i and B_j can be scheduled using the same model if the difference between $\omega(i)$ and $\omega(j)$ is less than the error in the query latency prediction model. By keeping a mapping of $\omega(x)$ to decision models⁵, WiSeDB can significantly reduce the amount of training that occurs during online scheduling.

Linear Shifting For some performance metrics, scheduling queries that have been waiting in the queue for a certain amount of time is the same as scheduling with a stricter deadline. This can significantly reduce the training overhead as the new models could reuse the scheduling graph of the previous ones as described in Section 5. Consider a performance goal that puts a deadline on each query to achieve a latency of three minutes. If a query is scheduled one minute after submission, that query can be scheduled as if it were scheduled immediately with a deadline of two minutes.

This optimization can be applied only to *linearly shiftable* performance metrics. In general, we say that a metric R is linearly shiftable if the penalty incurred under R for a schedule which starts queries after a delay of n seconds is the same as the penalty incurred under R' by a schedule that starts queries immediately, and where R' is a tightening of R by some known function of n . For the metric of maximum query latency in a workload, this function of n is the identity: the penalty incurred under some goal R after a delay of n seconds is equal to the penalty incurred under R' , where R' is R tightened by n seconds. For linear shiftable performance metrics, training a new model when new queries arrive can be reduced to shifting a decision model, as described in Section 5.

7. EXPERIMENTAL RESULTS

⁵ Experimentally, we found that this mapping can be stored using a few MB since each decision tree is relatively small.

Next we present our experimental results, which focus on evaluating WiSeDB’s *effectiveness* to learn low-cost workload schedules for a variety of performance goals, as well as its runtime *efficiency*.

7.1 Experimental Setup

We implemented WiSeDB using Java 8 and installed it on an Intel Xeon E5-2430 server with 32GB of RAM. The service generates scheduling strategies for a database application deployed on 6 Intel Xeon E5-2430 servers that can host up to 24 VMs with 8GB of RAM each. This private cloud emulates Amazon AWS [1] by using query latencies and VM start-up times measured on `t2.medium` EC2 instances. By default, our experiments assume a single type of VM unless otherwise specified.

Our database application stores a 10GB configuration of the TPC-H [4] benchmark on Postgres [3]. Query workloads consist of TPC-H templates 1 - 10. These templates have a response time ranging from 2 to 6 minutes, with an average latency of 4 minutes. We set the query processing cost to be the price of that instance ($f_r = \$0.052$ per hour), and we measured its start-up cost experimentally based on how long it took for the VM to become available via connections after it entered a “running” state ($f_s = \$0.0008$).

We trained our models on $N = 3000$ sample workloads with a size of $m = 18$ queries per workload, with queries executed in isolation. Higher values for N and m added significant training overhead without improving the effectiveness of our models while lower ones resulted in poor decision models. We generated our models using the J48 [5] decision tree algorithm and used them to generate schedules for incoming workloads. Our experiments vary the size of these workloads as well as the distribution of queries across the templates. Each experiment reports the average cost over 5 workloads of a given size and query distribution.

Performance Metrics. Our experiments are based on four performance goals: (1) **Max** requires the maximum query latency in the workload to be less than x minutes. By default, we set $x = 15$, which is 2.5 times the latency of the longest query in our workload. There is a charge of 1 cent per second for any query whose latency exceeds x minutes. (2) **PerQuery** requires that each query of a given template not exceed its expected latency by a factor of x . By default, we set $x = 3$ so that the average of these deadlines is approximately 15 minutes, which is 2.5 times the latency of the longest query. There is a charge of 1 cent per second in which any query exceeds x times its predicted latency. (3) **Average** requires that the average latency of a workload is x minutes. We set $x = 10$ so that this deadline is 2.5 times the average query template latency. There is a charge of a number cents equal to the difference between the average latency of the scheduled queries and x . (4) **Percent** requires that $y\%$ of the queries in the workload finish within x minutes. By default, we set $y = 90$ and $x = 10$. If $y\%$ of queries finish within x minutes, there is no penalty. Otherwise, a penalty of one cent per second is charged for time exceeding x minutes.

7.2 Effectiveness Results

To demonstrate the effectiveness and versatility of our approach, we compare schedules generated by WiSeDB with optimal schedules and known heuristics that have been shown to offer low-cost solutions for specific performance metrics.

Optimality Since our scheduling problem is computationally expensive, calculating an optimal schedule for large workloads is not feasible. However, for smaller workload sizes, we can exhaustively search for an optimal solution. Figure 9 shows the final cost for scheduling workloads of 30 queries uniformly distributed across 10 query templates. We compare schedules generated by WiSeDB

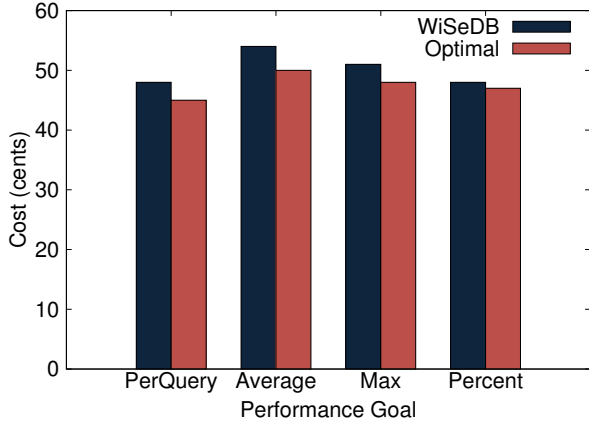


Figure 9: Optimality for various performance metrics

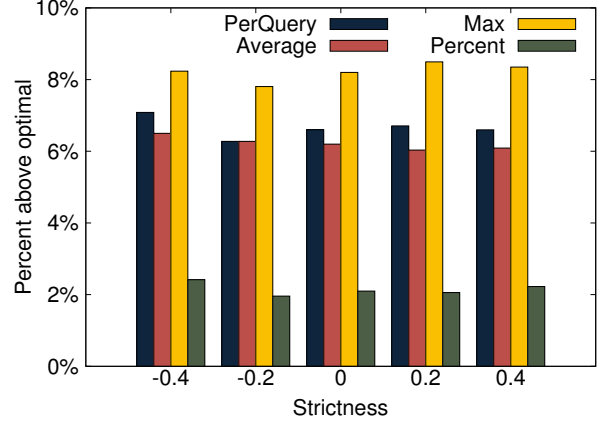


Figure 11: Optimality for varying constraints

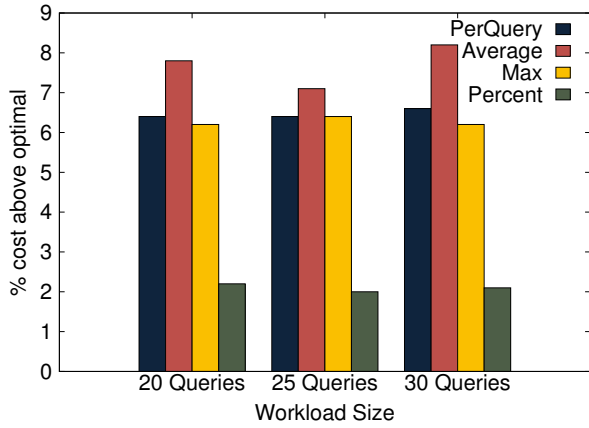


Figure 10: Optimality for varying workload sizes

with the optimal schedule (Optimal) for all our performance metrics. Our schedules are within 8% of the optimal for all metrics.

Figure 10 shows the percent-increase in cost over the optimal for three different workload sizes. WiSeDB consistently performs within 8% from the optimal independently of the size of the workload, while for the *Percent* metric, WiSeDB learns scheduling heuristics that cost less than 2% above the best solution.

Figure 11 shows the percent-increase in cost over the optimal when we tighten the constraints of the performance goals. To capture this parameter, we introduce a *strictness factor*. A strictness factor of 0 means a performance goal equal to those described in Section 7.1, whereas a strictness factor $x < 0$ represents a performance goal that is only x times as strict as those described above (so it has a more relaxed constraint). A strictness factor of $x > 0$ corresponds to a performance goal that is x times stricter than those above (so it has a tighter constraint). Figure 11 shows that the strictness factor does not affect the effectiveness of WiSeDB relative to the optimal. In the rest of the experiments, we will use performance goals with a strictness factor of 0.

We conclude that WiSeDB is able to learn effective strategies for scheduling incoming TPC-H workloads for various performance metrics independently of the size of the runtime workloads and the strictness of the performance goal.

Multiple VM Types We also trained decision models assuming the availability of multiple VM types. Figure 12 shows the cost of WiSeDB schedules against the optimal schedule with one

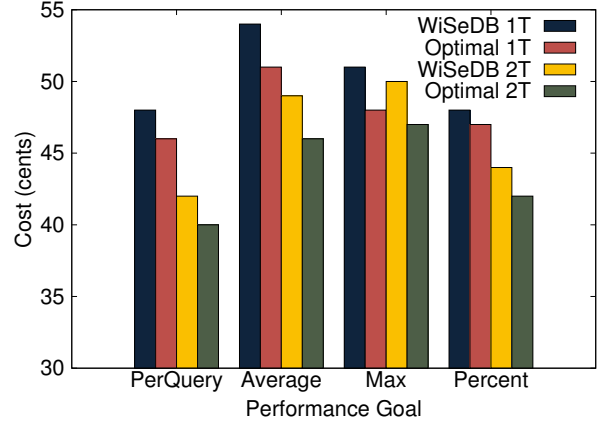


Figure 12: Optimality for multiple VM types

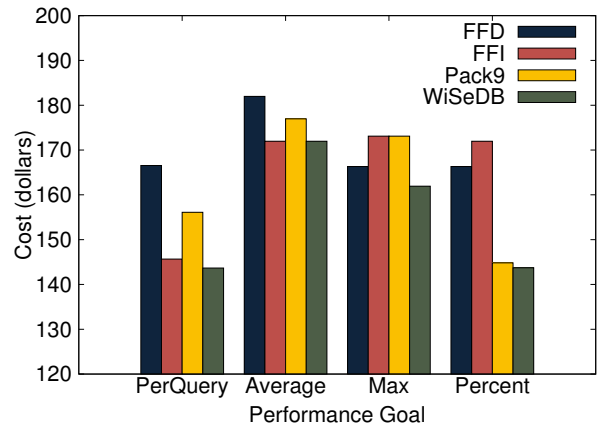


Figure 13: Comparison with metric-specific heuristics

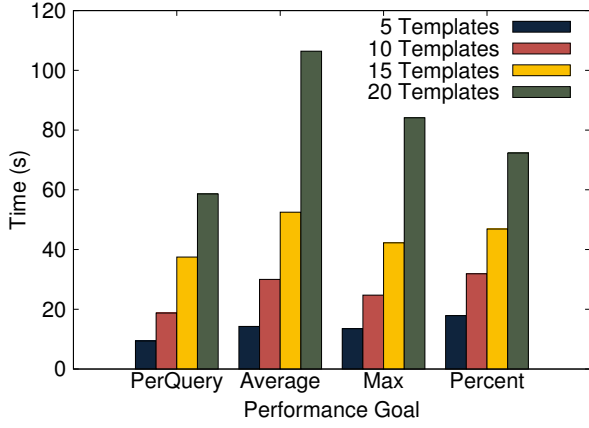


Figure 14: Training time vs. # of query templates

(WiSeDB 1T) and two (WiSeDB 2T) VM types. The first type is the `t2.medium` EC2 type and the second is the `t2.small` type. With the TPC-H benchmark, queries that require less RAM tend to have similar performance on `t2.medium` and `t2.small` EC2 instances. Since `t2.small` instances are cheaper, it makes sense to place low-RAM queries on `t2.small` instances. The results reveal that even when the learning task is more complex (using more VM types adds more edges to the scheduling graph and thus more complexity to the problem), WiSeDB is able to learn these correlations and generate schedules that perform within 6% of the optimal on average. Additionally, the performance always improved when the application was given access to a larger number of VM types. *Hence, WiSeDB is able to leverage the availability of various VM types, learn their impact on specific performance goals, and adapt its decision model to produce low-cost schedules.*

Metric-specific Heuristics We now evaluate WiSeDB’s effectiveness in scheduling large workloads of 5000 queries. Here, an exhaustive search for the optimal solution is infeasible, so we compare WiSeDB’s schedules with schedules produced by heuristics designed to offer good solutions for each performance goals. *First-Fit Decreasing (FFD)* sorts the queries in descending order and then places each query into the first VM with sufficient space. If no such VMs can be found, a new one is created. FFD is often used as a heuristic for classic bin-packing problems [22], indicating that it should perform well for the `Max` metric. *First-Fit Increasing (FFI)* does the same but first sorts the queries in ascending order, which works well for the `PerQuery` and the `Average` query latency metrics [28]. *Pack9* first sorts the queries in ascending order, then repeatedly places the 9 shortest remaining queries followed by the largest remaining query. Pack9 should perform well with the `Percent` performance goal because it will put as many of the most expensive queries into the 10% margin allowed.

Figure 13 shows the performance of WiSeDB compared to these approaches. The results show that there is no single simple heuristic that is sufficient to handle diverse performance goals. Our service offers schedules that consistently perform better than all other heuristics. *This indicates that WiSeDB outperforms standard metric-specific heuristics, i.e., its training features are effective in characterizing optimal decisions and learning special cases of query assignments and orderings with the VMs that fit better with each performance goal.* An example of such a special case was given in Section 4.5.

7.3 Efficiency Results

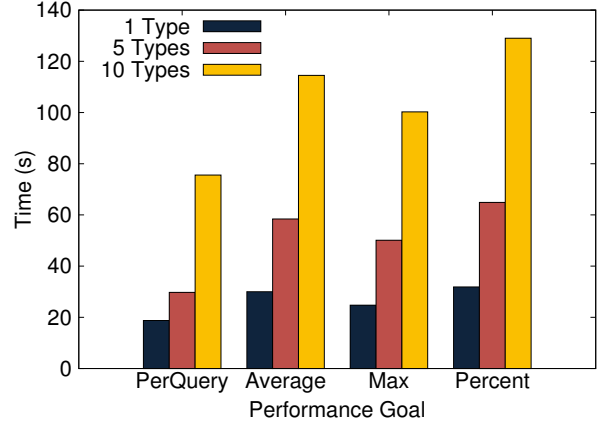


Figure 15: Training Time vs. # of VM types

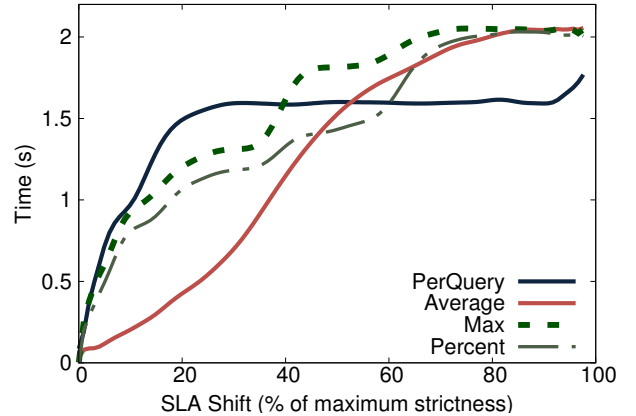


Figure 16: Overhead of adaptive modeling

Training Overhead WiSeDB trains its decision models offline. The training time depends on the number of templates in the workload specification as well as the number of different VM types available through the IaaS provider. Figure 14 shows how long it takes to train models for our four performance metrics, a single VM type, and varying numbers of query templates. For this experiment, we used additional query templates from the TPC-H benchmark. Here, the higher the number of query templates, the longer the training process since additional query templates represent additional edges that must be explored in the scheduling graph. In the most extreme cases, training can take around two minutes. In more tame cases, training takes less than 20 seconds. In Figure 15, we fix the number of query templates at 10 and vary the number of VM types available. Again, at the extreme ends we see training times of up to two minutes, with tamer multi-VM type cases taking only 30 seconds. *Hence, WiSeDB can learn metric-specific strategies in timely manner while each model needs to be trained once offline and can be applied to any number of incoming workloads.*

WiSeDB can adaptively train decision models by tightening the performance goal of its original model (Section 5). Figure 16 shows the retraining time when tightening the performance constraint by a factor of p . In general, we tighten a performance goal by a percentage p using the formula $t + (g - t) * (1 - p)$, where t is the strictest possible value for the performance metric, and g is the original constraint (described in Section 7.1). For example, the Max goal has an original deadline of 15 minutes, and, since the longest template in our workload specification is 6 minutes, the strictest possible deadline is 6 minutes. Tightening the Max goal by 33% means decreasing the deadline from 15 to 12 minutes.

Figure 16 shows that all four metrics can be tightened by up to 40% in less than a second. Tightening a constraint by a larger percentage takes more time since the number of training samples that have to be retrained increases. This is most immediately clear for the Max metric. The jump at 37% shift represents a large portion of the training set that needs to be recalculated. With tightening by only 20%, the optimal schedules used for training do not need to be modified, but a tightening by 40% causes more violations and hence new optimal schedules need to be re-calculated. PerQuery and Percent have curves that behave similarly.

The Average metric is slightly different. Since the query workloads are drawn uniformly from the set of query classes, their sum is normally distributed (central limit theorem). The average latency of each training sample is thus also normally distributed (since a normal distribution divided by a constant is still normal). Consider two tightenings of $X\%$ and $Y\%$, where $X \geq Y$. Any training sample that must be retrained when tightening by $Y\%$ will also need to be retrained when tightening by $X\%$. Each point on the Average curve in Figure 16 can be thought of as being the previous point plus a small δ , where δ is the number of additional samples to retrain. The curve is thus the sum of these δ s, and thus it approximates a Gaussian cumulative density function. *In most cases, WiSeDB generates a set of alternative models that explore the performance vs. cost trade-off though stricter or more relaxed performance goals in under a minute.*

7.4 Batch & Online Query Scheduling

Batch Scheduling WiSeDB’s models are used during runtime to generate schedules for incoming query batches. Section 7.2 discussed the effectiveness of these schedules. Figure 17 shows the time required to generate schedules for workloads of various sizes. WiSeDB scales linearly with the number of queries and it can schedule up to 30,000 queries in under 1.5 seconds. WiSeDB’s decision trees were usually shallow (height $h < 30$), so each de-

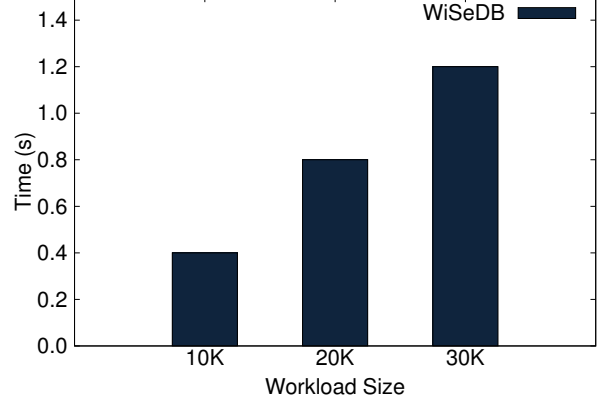


Figure 17: Scheduling overhead vs. batch size

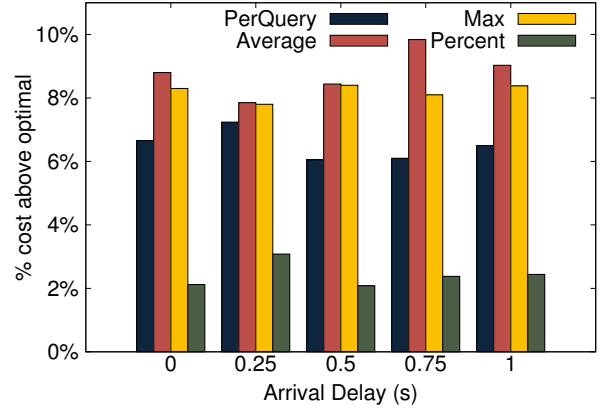


Figure 18: Effectiveness of online scheduling

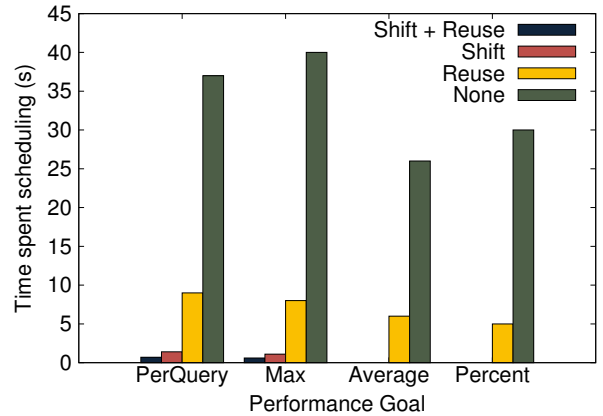


Figure 19: Average overhead for online scheduling

cision to assign a query or to provision a new VM can be made quickly. The runtime complexity of generating schedules using a decision model is bounded by $O(h \times n)$, where h is the height of the decision tree and n is the number of queries in our batch.

Systems or heuristics that must sort the queries in the batch beforehand, or systems that must scan over all provisioned VMs in order to make each query assignment or provisioning decision (e.g., [7, 29]), may not perform as well as WiSeDB for very large workloads. Since there must be at least one query on each VM, the maximum number of times the tree needs to be parsed is $2n$, where n is the number of queries in the batch. Each parse takes $O(h)$ time, so the runtime complexity of generating schedules using a decision model is bounded by $O(hn)$.

We note that the scalability of WiSeDB does not depend on the number of VMs (the number of VM to be rented is an output of our models). *We thus conclude that WiSeDB's decision models scale linearly and can be used to schedule efficiently very large batch workloads.*

Online Scheduling WiSeDB also supports online scheduling by training a new model with additional query templates when a new query arrives. Section 6.3 describes our approach as well as two optimizations for reducing the frequency of model re-training. The model reuse optimization (Reuse) can be applied to all four performance metrics. The linear shifting optimization (Shift) can only be used for the Max and PerQuery metrics.

Figure 18 shows the performance (cost) of WiSeDB compared to an optimal scheduler for various query arrival rates and performance metrics. For each metrics, we generate a set of 30 queries and run them in a random order, varying the time between queries. More time between queries means that the scheduler can use fewer parallel VMs, thus reducing cost. The results demonstrate that WiSeDB compares favorably with the optimal. In all cases, it generates schedules with costs that are within 10% of the optimal.

Figure 19 shows the impact of our optimizations on average scheduling overhead, i.e., the average time a query waits before being assigned to a VM. Here, Shift refers to using only the shifting optimization, Reuse refers to using only the model reuse optimization, and Shift+Reuse refers to using both optimizations. We compare these with None, which re-trains a new model at each query arrival. We use a query arrival rate that is normally distributed with a mean of $\frac{1}{4}$ seconds and standard deviation of $\frac{1}{8}$. If a query arrives before the last query was scheduled, it waits.

Figure 19 shows that the average query wait time can be reduced to below a second for the PerQuery and Max performance goals using both the linear shifting and the model reuse optimization. The Average and Percent performance goals have substantially longer wait times, at around 5 seconds, but a 5 second delay represents only a 2% slowdown for the average query. *Hence, our optimizations are very effective in reducing the query wait time, allowing WiSeDB to efficiently reuse its generated models and offer online scheduling solutions in a timely manner.*

7.5 Sensitivity Analysis

Skewed Workloads We also experimented with scheduling workloads heavily skewed towards some query templates. Figure 20 shows the average percent increase over the optimal cost for a workload with varying χ^2 statistics [13], which indicates the skewness factor. The χ^2 statistic measures the likelihood that a distribution of queries was not drawn randomly: 0 represents a uniform distribution of queries w.r.t. templates, and 1 represents the most skewed distribution possible, i.e., a batch which includes only a single template (while the decision model is trained for workloads of 10 templates). The χ^2 statistics were calculated with the null

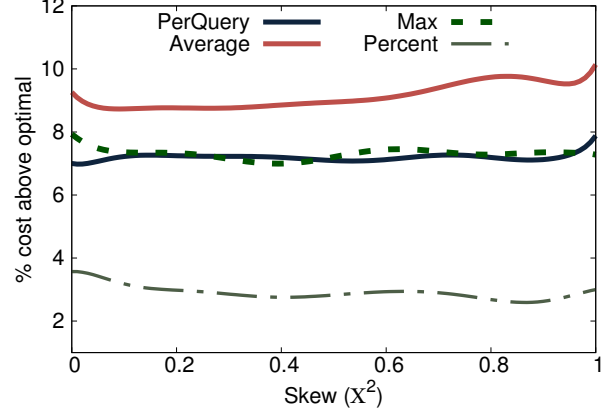


Figure 20: Sensitivity to skewed runtime workloads

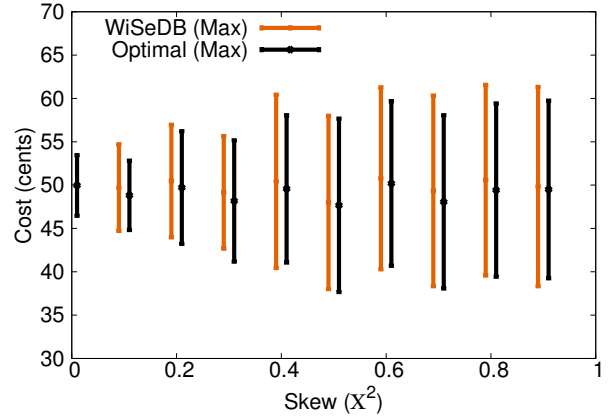


Figure 21: Workload skewness vs. cost range

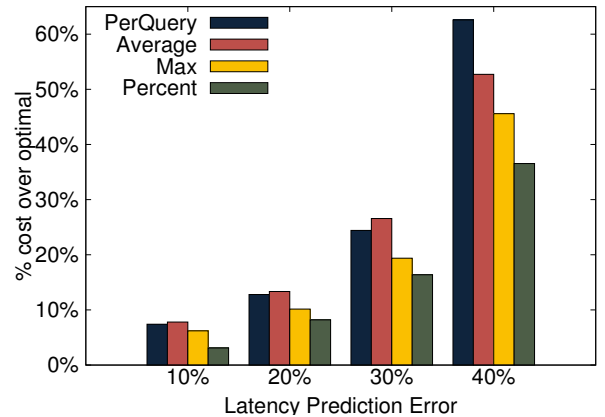


Figure 22: Optimality for varying cost errors

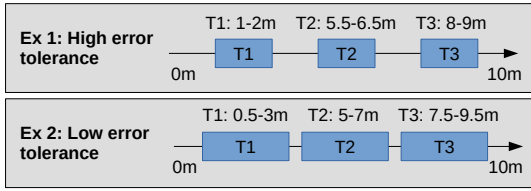


Figure 23: Example of templates with low and high error tolerance

hypothesis that each query template would be equally represented. Thus, the value on the x-axis is the confidence with which that hypothesis can be rejected. Even with highly skewed workloads consisting of almost exclusively a single query template ($\chi^2 \approx 1$), the average percent-increase over the optimal changes by less than 2%.

To better understand these results, we used WiSeDB to schedule 1000 workloads (instead of our default 5 workloads) under different skewness factors for the `Max` metric. Figure 21 shows both the average and the range of the cost of these schedules. While the mean cost remains relatively constant across different χ^2 values (as in Figure 20), the variance of the cost increases as skew increases. This is because a very skewed workload could contain a disproportionate number of cheap or expensive queries, whereas a more uniform workload will contain approximately equal numbers of each. WiSeDB’s decision models have variance approximately equal to that of an optimal scheduler. *Hence, WiSeDB’s models perform effectively even in the presence of skewed query workloads.*

Latency Prediction Accuracy WiSeDB relies on a query latency prediction model to estimate penalty costs. While existing query performance prediction models [10, 11] can be used, these often exhibit prediction errors, leading to incorrect estimations of the penalty costs. Naturally, we expect that the effectiveness of schedules produced by WiSeDB to decrease as the cost model error increases, but we discovered that WiSeDB is able to tolerate a certain level of prediction errors. *In fact, the more distinguishable templates are with respect to their latency (see Figure 23), the higher prediction error WiSeDB can tolerate.* This is because WiSeDB expects queries with similar latencies to be assigned to the same template. Therefore, the presence of latency prediction errors causes some queries to have ambiguous template membership, hindering WiSeDB’s ability to learn effective models.

Figure 22 demonstrates this conclusion. Here, each σ^2 value refers to the cost model error (standard deviation) as a percentage of the actual query latency. WiSeDB handles cost model errors less than 30% very well. This is because, given our template definitions, the percentage of queries who are assigned to the wrong template at 30% error, e.g. a query with actual latency similar to template T_i is mistakenly assigned to template T_j , is 14%. At 40% error, this percentage rises to 67%, leading to poor scheduling decisions.

8. RELATED WORK

Many research efforts address various aspects of workload management in cloud databases. iCBS [8] offers a generalized profit-aware heuristic for ordering queries, but the algorithm considers assignments to a single VM, and performance goals are limited to piecewise-linear functions (which cannot express percentile performance metrics). In [9], they propose a data structure to support profit-oriented workload allocation decisions, including scheduling and provisioning. However, their work supports only step-wise SLAs, which cannot express average or percentile goals. In [12, 18, 19, 20], they consider the problem of mapping each tenant (customer) to a single physical machine to meet performance goals, but ordering and scheduling queries is left to the application. SmartSLA [29] offers a dynamic resource allocation approach for

multi-tenant databases. WiSeDB supports a wider range of metrics than the above systems and its decision models offers holistic solutions that indicate the VMs to provision, query assignments, and query execution order. WiSeDB also *learns* decision models for various SLA types, as opposed to utilizing a hand-written, human-tailored heuristic. This brings an advantage of increased flexibility (changing performance goals without reimplementing) at the cost of some training overhead.

In [27, 30], they propose admission control solutions that reject queries that might cause an SLA violation at runtime, whereas our system seeks to minimize the cost of scheduling every query and to inform the user of performance/cost trade-offs. Even if a query cannot be processed profitably, WiSeDB will still attempt to place it in a cost-minimal way. [24] proposes multiple SLAs with different prices for various query workloads, but leaves query scheduling up to the application and supports only per-query latency SLAs, whereas we allow applications to define their own query and workload-level performance metrics.

In [23], they propose monitoring mechanism for resource-level SLAs, and in [16], they propose an approach for translating query-level performance goals to resource requirements, but both assume only a single performance metric and leave query scheduling up to the application. WiSeDB takes a query-centric as opposed to a resource-centric approach, assuming that a latency prediction model can correctly account of resource oscillations and query affinity. [26] proposes an end-to-end cloud infrastructure management tool, but it supports only constraints on the maximum query latency within a workload and uses a simple heuristic for task scheduling. WiSeDB learns specific heuristics for application-specific performance goal by training on optimal schedules for representative workload samples. In [7], they use a hypergraph partitioning approach to schedule tasks expressed as directed acyclic graphs on cloud infrastructures. While WiSeDB contains no notion of query dependency, [7] does not consider performance goals of any type, nor provisioning additional resources. Finally, [15] proposes a system for scheduling Hadoop jobs on clouds which takes resource provisioning and deadlines into account. However, it considers only simple per-task deadlines and is not easily generalized beyond Hadoop.

9. CONCLUSIONS

This work introduces WiSeDB, a workload management advisor for cloud databases. To the best of our knowledge, WiSeDB is the first system to address workload management in an *holistic* fashion, handling the tasks of resource provisioning, query placement, and query scheduling for a broad range of performance metrics. WiSeDB leverages machine learning techniques to learn decision models for guiding the low-cost execution of incoming queries under application-defined performance goals. We have shown that these decision models can efficiently and effectively schedule a wide range of workloads. These models can be quickly adapted to enable exploration of the performance vs. cost trade-offs inherent in cloud computing, as well as provide online query scheduling with little overhead. Our experiments demonstrate that WiSeDB can gracefully adapt to errors in cost prediction models, take advantage of multiple VM types, process skewed workloads, and outperform several well-known heuristics with small training overhead.

We have a full research agenda moving forward. We are currently investigating alternative features for characterizing the optimal assignment decision as well as alternative learning techniques (e.g., neural networks, reinforcement learning) for the workload management problem. We are also looking into multi-metric performance goals that combine workload and query level constraints,

as well as dynamic constraints that change based on some external variable, e.g. time of day.

10. ACKNOWLEDGMENTS

This research was funded by NSF IIS 1253196.

11. REFERENCES

- [1] Amazon Web Services, <http://aws.amazon.com/>.
- [2] Microsoft Azure Services, <http://www.microsoft.com/azure/>.
- [3] PostgreSQL database, <http://www.postgresql.org/>.
- [4] The TPC-H benchmark, <http://www.tpc.org/tpch/>.
- [5] Weka 3, <http://cs.waikato.ac.nz/ml/weka/>.
- [6] K. D. Ba, H. L. Nguyen, H. N. Nguyen, and R. Rubinfeld. Sublinear time algorithms for earth mover's distance. *Theory of Computing Systems*, 48(2):428–442, 2011.
- [7] U. V. Catalyurek, K. Kaya, and B. Ucar. Integrated data placement and task assignment for scientific workflows in clouds. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 45–54. ACM, 2011.
- [8] Y. Chi, H. J. Moon, and H. Hacigumus. iCBS: Incremental cost-based scheduling under piecewise linear SLAs. In *Proceedings of the VLDB Endowment*, volume 4.9 of *VLDB '11*, pages 563–574. VLDB Endowment, June 2011.
- [9] Y. Chi, H. J. Moon, H. Hacigumus, and J. Tatemura. SLA-tree: A framework for efficiently supporting SLA-based decisions in cloud computing. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '11, pages 129–140. ACM, 2011.
- [10] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 337–348. ACM, 2011.
- [11] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal. Contender: A resource modeling approach for concurrent query performance prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '14, pages 109–120, 2014.
- [12] A. J. Elmore, S. Das, A. Pucher, D. Agrawa, A. E. Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 517–528. ACM, 2013.
- [13] P. E. Greenwood and M. S. Nikulin. *A guide to chi-squared testing*, volume 280. John Wiley & Sons, 1996.
- [14] P. Hart, N.J. Nilsson, and R. B. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.
- [15] E. Hwang and K. H. Kim. Minimizing cost of virtual machines for deadline-constrained MapReduce applications in the cloud. In *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, GRID '12, pages 130–138, Sept 2012.
- [16] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14. ACM, 2012.
- [17] S. Koenig and M. Likhachev. A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the International Conference on Automated Planning and Scheduling*, ICAPS '06, pages 402–405, 2006.
- [18] W. Lang, S. Shankar, J. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. In *Knowledge and Data Engineering, IEEE Transactions on*, volume 26.6 of *ICDE '14*, pages 1447–1463, June 2014.
- [19] Z. Liu, H. Hacigumus, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 442–453. ACM, 2013.
- [20] H. Mahmoud, H. J. Moon, Y. Chi, H. Hacigumus, D. Agrawal, and A. El-Abbadi. CloudOptimizer: Multi-tenancy for I/O-bound OLAP workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 77–88. ACM, 2013.
- [21] R. Marcus and O. Papaemmanouil. Workload management for cloud databases via machine learning. In *Workshop on Cloud Data Management and the IEEE International Conference on Data Engineering*, CloudDM '16. IEEE, 2016.
- [22] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [23] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: Performance isolation in multi-tenant relational database-as-a-service. In *6th Biennial Conference on Innovative Data Systems Research*, CIDR '13, January 2013.
- [24] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *Conference on Innovative Data Systems Research*, CIDR '15, 2015.
- [25] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Records*, 29(4):64–71, Dec. 2000.
- [26] B. Sotomayor, R. Montero, I. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, Sept 2009.
- [27] S. Tozer, T. Brecht, and A. Aboulhaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, ICDE '10, pages 397–408, March 2010.
- [28] V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.
- [29] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, ICDE '11, pages 87–98. IEEE, 2011.
- [30] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigumus. ActiveSLA: A profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 15:1–15:14. ACM, 2011.